"Express Mail" mailing label number:

EL684226268US

DYNAMIC COMPILATION CONTROL

David M. Ungar

BACKGROUND

Field

5   [1001]   The invention relates to selective optimization of portions of a computer program based on observations or predictions of program execution behavior, in particular, to techniques for dynamically optimizing selected portions of a computer program based on identification of a break-even point.

Description of the Related Art

10   [1002]   Typically, software programmers generate software programs by writing language statements in a human-readable language to create a file containing source statements or source code. In order to run the file, the source statements must be prepared for execution by being made machine-readable. For example, appropriate language compiler is typically run on file containing the source statements. A

15   compiler is a special program that processes statements written in a particular programming or human-readable language and turns them into machine-readable language or code that is usable by a computer's processing unit. When executing or running, the compiler operates on the human-readable program by first parsing or analyzing all of the language statements and then, in one or more successive stages or

20   passes, builds the output code. Traditionally, the output of the compilation has been called object code or sometimes an object module. The object code is machine code that a processing unit can process or execute one instruction at a time.

[1003]   The Java™ programming language, a language used in object-oriented programming, has popularized the possibility of compiling output (e.g., "bytecode")

25   that can run on any computer system platform for which a Java virtual machine or bytecode interpreter is provided to convert the bytecode into instructions that can be

- 1 -

executed by the actual hardware of the platform. A bytecode interpreter, or simply "interpreter," will interpret the bytecode one instruction at a time for execution by the hardware. Alternatively, a Java just-in-time ("JIT") compiler (e.g., a second compiler) at the particular system platform compiles the bytecode into the particular

5   system code as though the program had been compiled initially on that platform. Although recompiling a bytecode often takes more time than interpreting the bytecode, once the code has been recompiled by the JIT compiler it will usually run more quickly on the hardware platform. Thus, an interpreter favors startup over execution efficiency while a compiler slows start-up time but executes programs

10   faster. Notwithstanding, a typical JIT compiles all code that is executed, and hence must attain compilation speed by sacrificing code quality. Thus, source code may be compiled into bytecode which is readable by a Java virtual machine, and the bytecode may in turn be recompiled or interpreted for execution by the actual hardware.

[1004]   The Java programming language has stimulated work on high performance

15   implementations that dynamically compile or optimize frequently executed portions of programs. The common practice of compiling everything that executes even one time can squander compilation time. Systems that deviate from this practice typically rely on ad hoc heuristics. A reasoned and effective set of heuristics is desirable to optimize the effect of such dynamic compilation, so that time is spent optimally

20   compiling program portions that are frequently executed and not wasted compiling infrequently used program portions.

SUMMARY

[1005]   It has been discovered that certain portions of a computer program may be effectively selected for interpretation or recompilation using a compilation threshold

25   which is calculated using an execution break-even value or point, and a program portion hazard rate. A break-even value provides the number of expected portion executions at which it becomes cost effective to recompile and optimize the program portion for future executions rather than interpret the portion immediately prior to each expected execution. With perfect knowledge of the future execution of a

30   program, this value provides a useful compilation threshold value. However, such perfect knowledge is often unavailable. When there is imperfect knowledge regarding

- 2 -

the execution of a program, principles of reliability engineering can be used to calculate a program portion hazard rate for use in providing a more effective compilation threshold. The program portion hazard rate provides the probability that a program portion will stop being executed given that the program portion has already

5   been executed a given number of times. An effective compilation threshold may be set to correspond to the point at which at which a decreasing hazard rate corresponds to a reciprocal of the break-even value.

[1006]   By observing execution properties of portions of a computer program and by applying such a computed compilation threshold, individual portions of the

10   computer program may be selectively recompiled during execution of the computer program based on correspondence between the observed execution for the individual portions and the compilation threshold. Such a system and method can be used to optimize dynamic compilation so that time is spent optimally compiling program portions that are determined to be frequently executed going forward and not wasted

15   compiling program portions that are determined to be infrequently executed going forward.

[1007]   Thus, in one embodiment of the present invention, a method of selecting certain portions of a computer program for compilation is provided. The method includes computing a compilation threshold. The compilation threshold corresponds

20   to a point at which a decreasing hazard rate of a program portion corresponds to the reciprocal of a break-even number of executions of the program portion that recoup computational costs of compilation. During execution of the computer program, the program portions are dynamically compiled based on correspondence between observed execution for the individual portions and the compilation threshold. The

25   computation of the compilation threshold may be performed coincident with execution of the computer program, and may use runtime information from the execution. Also, the hazard rate, $hr(x)$, for a particular one of the computer program portions may be used to at least approximate a probability that the particular program portion will stop being executed in the computer program given that the particular

30   portion has executed x times. In a further embodiment, the break-even number of executions may be defined to be a function of time or other execution resource to (i) execute an uncompiled version of a particular one of the computer program portions,

(ii) compile the particular portion, and/or (iii) execute a compiled version of the particular portion.

[1008]   In another embodiment, an execution environment is provided for a computer program encoded using execution codes that may optionally be executed in either first or second form. The execution environment includes a dynamic compilation mechanism that transforms an implementation of a particular execution code to the second form thereof, wherein the second form is substantially optimized as compared to the first form. For at least the particular execution code, the execution environment includes an execution-time measurement of execution frequency at which a decreasing hazard rate corresponds to a reciprocal of a break-even number of executions that recoup computational costs of transformation to the second form. The dynamic compilation mechanism is responsive to the execution-time measurement. In a further embodiment, the first and second forms respectively include uncompiled and compiled version of the execution code, or the first and second forms both include compiled versions of the execution code, but the second form is substantially optimized as compared to the first form. In yet a further embodiment, the execution code may be a bytecode.

In another embodiment, a computer program product is encoded in at least one computer readable medium. The computer program product includes first and second instructions executable in an information processing system. The first instructions are executable to instrument execution of a computer program executing thereon. The first instructions providing data indicative of execution frequency for at least a particular portion of the computer program. The second instructions are executable to identify a particular point in the execution of the computer program at which a decreasing hazard rate calculated from the execution frequency data for the particular portion of the computer program corresponds to a reciprocal of a break-even number of executions thereof that recoup computational costs of transformation to an optimized form.

[1009]   In a further embodiment, the particular portion is dynamically compiled to the optimized form coincident with identification of the particular point. In yet a further embodiment, the computer program product includes a dynamic compiler. In

- 4 -

yet a further embodiment, the computer program product is embodied as part of an execution environment for the computer program. In a further embodiment, the computer program product is embodied as part of the computer program. In a further embodiment, the at least one computer readable medium is selected from the set of a

5   disk, tape or other magnetic, optical, or electronic storage medium and a network, wireline, wireless or other communications medium.

[1010]   The foregoing is a summary and thus contains, by necessity, simplifications, generalizations and omissions of detail. Consequently, those skilled in the art will appreciate that the foregoing summary is illustrative only and that it is not intended to

10   be in any way limiting of the invention. Other aspects, inventive features, and advantages of the present invention, as defined solely by the claims, will become apparent in the detailed description set forth below.

BRIEF DESCRIPTION OF THE DRAWINGS

[1011]   The present invention may be better understood, and its numerous objects,

15   features, and advantages made apparent to those skilled in the art by referencing the accompanying drawings. The use of the same reference symbols in different drawings indicates similar or identical items.

[1012]   Figure 1 is a flow chart showing one embodiment of a method of running a computer program in accordance with the invention.

20   [1013]   Figure 2 is an information processing system structured in accordance with the invention.

[1014]   Figures 3A-3D show a series of plots illustrating the performance of an exemplary program in a system in accordance with the invention.

[1015]   Figures 4A-4E show a series of plots illustrating the performance of an

25   exemplary program in a system in accordance with the invention.

DESCRIPTION OF THE PREFERRED EMBODIMENT(S)

[1016]   The following discussion is intended to provide a detailed description of at least one example of the invention to aid those skilled in the art to understand and

practice the invention. For example, Figure 1 shows an exemplary information processing system which provides a useful reference in discussing various aspects of the invention. Of course, the information processing system has been simplified for purposes of discussion, and it is just one of may different types of appropriate systems

5   that may be used in accordance with the invention. Consequently, the following discussion should not be taken to be limiting of the invention itself. Rather, any number of variations may fall within the scope of the invention which is properly defined in the claims following this description.

[1017]   Referring to Figure 1, system 100 includes storage 110, preparation unit

10   120, and execution unit 180. Storage 110 includes program 140. Storage 110 is coupled to provide program 140 to preparation unit 120 via connection 142. Preparation unit 120 is coupled to provide compiled and/or interpreted program portions to execution unit 180 via connection 182. As mentioned, the architecture shown in Figure 1 is simplified for the purposes of discussion. For example,

15   connections described herein may be physical or logical, and may be a part of a path including other elements which are not shown so as to avoid obfuscation of the invention. One of ordinary skill in the art will realize, based on the teachings herein, that the operations, signals and connections discussed below in relation to system 100 may be performed under control or by way of other logic blocks or collections of

20   other logic blocks which may include hardware and/or software, integrated and/or distributed.

[1018]   Storage 110 is any one of a variety of computer readable media appropriate for storing a computer program such as program 140. Program 140 is any computer program including at least some source code or other code which must be compiled

25   and/or interpreted prior to use by execution unit 180. For example, program 140 may be a Java program, and may or may not already be compiled into bytecode. Storage 110 is coupled to provide portions of program 140 to preparation unit 120 via connection 142. Such portions may correspond to functions, procedures, methods, routines, objects or bytecodes of program 140, to name just a few examples. Storage

30   110 may also include various control hardware and software (not shown).

[1019]   Preparation unit 120 includes dynamic compilation controller 150 which is coupled to receive those portions of program 140 which are to be executed by execution unit 180 and/or to receive information regarding those portions. As with other architectural blocks of system 100, dynamic compilation controller 150 may

5   include hardware and software of one or several other architectural blocks. Dynamic compilation controller 150 may also include or operate under control of portions of program 140 or other software stored in storage 110.

[1020]   Preparation unit also includes interpreter 160 and compiler 170, which are coupled to receive the portions under the control of dynamic compilation controller

10   150 via connection 152 and connection 154, respectively. Using compilers 160, 170, preparation unit 120 prepares (e.g., compiles and/or interprets) the received portions of program 140 so that such portions will be executable by execution unit 180. Thus, in the present embodiment, preparation unit 120 is capable, among other things, of providing a just-in-time compile of program 140 for execution unit 180. Preparation

15   unit 120 provides system 100 with virtual machine capability.

[1021]   Dynamic compilation controller 150 routes portions to one of interpreter 160 and compiler 170 for preparation for execution by execution unit 180. To control this routing, dynamic compilation controller 150 analyzes a break-even value, a compilation threshold, and an observed execution frequency or count, for the program

20   or portions thereof. Such analysis is described in greater detail below with reference to Figure 2. Dynamic compilation controller 150 determines whether the compilation threshold has been reached for each program portion received from storage 110. If the compilation threshold has not been reached for a particular program portion, dynamic compilation controller 150 routes such program portion to interpreter 160

25   where the program portion is interpreted or quickly compiled for execution by execution unit 180. If the compilation threshold has been reached for a particular program portion, dynamic compilation controller 150 routes the program portion to compiler 170 for recompiling or optimization of the program portion for execution by execution unit 180. Recompilation by compiler 170 is often more costly in terms of

30   time or some other system resource measure than is interpretation by interpreter 160, but recompilation is typically more effective than interpretation in reducing execution time by execution unit 180.

- 7 -

[1022]   Execution unit 180 receives prepared (e.g., interpreted and/or compiled) portions from one of interpreter 160 and compiler 170, and carries out the functionality of such portions. If portions have been compiled by compiler 170, such compiled portions may be temporarily stored (e.g., in storage 110) for later executions of the such portions. Execution unit 180 may include hardware, firmware or software or a combination thereof. In one embodiment, preparation unit 120 and execution unit 180 may be individually viewed as a virtual machine and a microprocessor, respectively, and preparation unit 120 and execution unit 180 may be collectively viewed as a logical processor.

[1023]   Figure 2 shows the flow of execution of a computer program in accordance with one embodiment of the invention. In the embodiment shown, a program is executed and monitored to collect data regarding execution of the portions thereof. The execution data is used to determine a dynamic compilation threshold. Although the execution flow of Figure 2 is discussed below with reference to system 100 of Figure 1, the execution flow may be used in other exemplary systems.

[1024]   During set break-even value operation 210, a break-even value is set for the computer program or, alternatively, for each program portion. The break-even value is the number of expected portion executions at which it becomes cost effective to recompile and optimize the program portion for all or a substantial number of later executions rather than to interpret the portion immediately prior to each execution thereof. If a program portion is expected to be executed more than the break-even number of times, then it is more efficient to optimally compile the program portion even when such optimal compilation requires more time than interpretation of the program portion. If the expected number of executions of the program portion is less than the break-even value, then it is more efficient to interpret the program portion immediately prior to each future execution of the program portion. In one embodiment, the set break-even value step 210 may occur during execution of the program.

[1025]   In the embodiment shown, after set break-even value step 210 and during initiate program step 215, execution of program 140 on information processing system 100 is initiated. For example, a logic block of system 100 may call program

- 8 -

140 from storage 110, which provides program 140 to preparation unit 120 for preparation to be executed by execution unit 480.

[1026] After initiate program step 215 and during monitor program step 220, the execution of program 140 is monitored by system 100 (e.g., through dynamic compilation controller 150 and, if appropriate, other logic blocks including perhaps storage 110). For example, in one embodiment the frequency of executions of each program portion of program 140 is monitored by system 100.

[1027] After monitor program step 220 and during calculate hazard rate step 225, the hazard rate is calculated for each portion of program 140. The hazard rate as the probability that a bytecode will stop being executed, given that it has already run x times. It makes sense that when the hazard rate is too high we don't want to compile the bytecode because it is too likely to stop being executed. Likewise, when the hazard rate is too low, we are too late in compiling the bytecode, it is so unlikely to stop being executed that it should have been compiled before The theoretical and practical aspects of calculation and use of the hazard rate are discussed below in greater detail with reference to the provided analytical model.

[1028] After calculate hazard rate step 225 and during compilation threshold decision 230, dynamic compilation controller 150 determines whether the compilation threshold has been reached for each portion of program 140. As is discussed in greater detail below, the compilation threshold is triggered, if at all, when the decreasing hazard rate corresponds to the reciprocal of the break-even value. If the compilation threshold has not been reached, the portion is interpreted during interpret portion step 235 by interpreter 160. If the compilation threshold has been reached, the portion is recompiled during compile portion step 240 by compiler 170. The results of such compilation may be stored in storage 110 for subsequent executions of that portion by execution 180.

[1029] After interpretation or compilation by interpreter 160 and compiler 170 during steps 235 and 240, execution unit 180 executes the program portion during execute portion step 250. After execute portion step 250 and during last portion decision 260, system 100 determines whether the current program portion is the last program portion of program 140. If the current program portion is the last program

portion, execution of program 140 is terminated. If the current program portion is not the last program portion, control transitions to monitor program step 220.

Analytical Model

[1030]   An analytical model supporting the above embodiments has been

5   formulated that expresses the performance of such a system and that may be used to improve such performance. The model is based in part on a frequency histogram of bytecode execution. For a given program, a bytecode frequency histogram indicates how many times each bytecode is executed. Based on the teachings herein, the bytecode frequency histogram may be used to select an optimized (e.g., improved)

10   compilation threshold where the hazard rate falls through the reciprocal of the break-even value, the number of times a compiled bytecode must be executed to recoup its compilation time. The analytical model can help engineers and programmers to understand the consequences of simple policies, of compiler-design tradeoffs, and of different types of workloads, and it can suggest values for key parameters that are

15   provably optimal.

[1031]   In one embodiment, program 140 runs for a finite time, and it is desirable to minimize that time. Other embodiments may involve the optimization of a program based on pause time or interactive performance. In the above discussed example, code may be run in two ways: interpreted or compiled. If compiled, the compilation

20   occurs at run-time and so detracts from minimizing total running time. For the sake of simplicity, the following discussion focuses bytecodes as the program portions. That is, bytecodes are the units of both compilation and execution. Other embodiments may use other forms of portions.

[1032]   A few definitions will now be introduced. To introduce the analytical

25   model in the simplest way, one may assume for the sake of the immediately following discussion (and revisit later) that each bytecode takes the same amount of time to interpret, that each bytecode takes the same amount of time to compile, and that once compiled, each bytecode takes the same time to execute:

$T_I \equiv$ time to interpret a bytecode

30          $T_C \equiv$ time to compile a bytecode

$T_E \equiv$ time to execute a (compiled) bytecode

- 10 -

[1033]   Our analysis will be based on bytecode frequencies. That is, the analysis will be based not the relative frequencies of different kinds of bytecodes, but rather the number of bytecodes contained in the program that run a given number of times in the course of one run of a program:

5
$$f(x) \equiv \text{ the number of bytecodes that run x times}$$

[1034]   The (complementary) distribution as well as the density are usefully defined:

$$\overline{F}(x) \equiv \text{ the number of bytecodes that run at least x times}$$

$$\overline{F}(x) \equiv \int_x^\infty f(y)dy$$

10
$$\frac{d}{dx}\overline{F}(x) = -f(x)$$

$$\overline{F}(0) = \text{ the total number of bytecodes in the program}$$

[1035]   Note that another embodiment of the model would use summations instead of integrals because a bytecode cannot execute half a time, but the common statistical practice of approximating discrete functions with continuous ones is followed herein.

15   [1036]   The number of executions of bytecodes that run at least a certain number of times:

$$E(x) \equiv \# \text{ of executions of bytecodes that run at least x times}$$

$$E(x) = \int_x^\infty yf(y)\,dy$$

$$\frac{d}{dx}E(x) = -xf(x)$$

20
$$E(0) = \text{total} \# \text{ of executions of bytecodes in the program}$$

$$E(0) - E(x) = \# \text{ of executions of bytecodes that run less than x times}$$

[1037]   In the following discussion regarding the analytical model, a mathematical model for two cases is presented: one with perfect knowledge in which a program's

25   profile is known *a priori*, and one with imperfect knowledge in which the profile is unknown.

Compilation Policy in an Idealized Scenario: Perfect Knowledge

[1038]   Suppose that each program came with a profile that indicated how many times each of its bytecodes would run. Then, the obvious compilation policy would

- 11 -

be to first compile every bytecode that was going to run more than a certain number of times, and then to run the program, interpreting the infrequently running bytecodes and executing the (compiled) frequently running bytecodes. What would an optimal compilation threshold in this case be?

5 [1039] We start with

$$T_{tot}(x) \equiv \text{Total time to run the program, with compilation threshold x}$$

[1040] This time will be the sum of three components: the time to compile the compiled bytecodes, the time to execute the compiled bytecodes, and the time to interpret the non-compiled bytecodes. So

10
$$T_{tot}(x) = T_C \overline{F}(x) + T_E E(x) + T_I [E(0) - E(x)]$$
$$= T_I E(0) - [(T_I - T_E)E(x) - T_C \overline{F}(x)]$$

[1041] The second line above rearranges the total time to be a sum of the time it would take to interpret all executions, $T_I E(0)$, minus the net time saved by the compiler, $(T_I - T_E)E(x) - T_C \overline{F}(x)$. Rather than minimizing the total time, it will prove more convenient to maximize this latter term. We normalize it with respect to

15 $T_I - T_E$, the execution time saved by one compiled bytecode execution:

$$Speedup(x) \equiv \frac{T_I E(0) - T_{tot}(x)}{T_I - T_E}$$
$$= E(x) - \frac{T_C}{T_I - T_E} \overline{F}(x)$$

[1042] To find the optimal compilation threshold, we set the first derivative of the

20 speedup to zero (using our definitions for $E(x)$ and $\overline{F}(x)$ above), and will check for a negative second derivative:

$$\frac{d}{dx} Speedup(x) = -xf(x) + \frac{T_C}{T_I - T_E} f(x)$$
$$= \left(\frac{T_C}{T_I - T_E} - x \right) f(x)$$

[1043] Setting to zero:

25
$$\frac{d}{dx} T_{tot}(x) \bigg|_{x = x_{opt}} = \left(\frac{T_C}{T_I - T_E} - x_{opt} \right) f(x_{opt})$$

- 12 -

$$\therefore x_{opt} = \frac{T_C}{T_I - T_E} \quad or \quad f(x_{opt}) = 0$$

[1044]   and checking the second derivative:

$$\frac{d^2}{dx^2} T_{tot}(x) = \left(\frac{T_C}{T_I - T_E} - x \right) f'(x) - f(x)$$

$$0 > \frac{d^2}{dx^2} T_{tot}(x) \Big|_{x=x_{opt}}$$

$$0 > \left(\frac{T_C}{T_I - T_E} - x_{opt} \right) f'(x_{opt}) - f(x_{opt})$$

$$f(x_{opt}) > \left(\frac{T_C}{T_I - T_E} - x_{opt} \right) f'(x_{opt})$$

[1045]   This inequality shows that of the two possibilities for a zero first

derivative, $x_{opt} = \frac{T_C}{T_I - T_E}$, or $f(x_{opt}) = 0$, only the former condition yields an optimal

speedup. Thus,

$$x_{opt} = \frac{T_C}{T_I - T_E}$$

[1046]   Note that the optimal threshold does not depend on the frequency

distribution of the program. It is simply the ratio of the compilation time per bytecode

to the time saved by executing a bytecode instead of interpreting it. In other words, it

represents a break-even point; if this value is 10, then a compiled bytecode must

execute ten times in order to recoup the investment in compilation time. It is

convenient to denote this quantity with a new symbol:

$$\beta \equiv the\ break - even\ frequency \equiv \frac{T_C}{T_I - T_E} \quad so$$

$$Speedup(x) = E(x) - \beta \overline{F}(x) \quad and \quad x_{opt} = \beta$$

[1047]   Suppose a very fast (and stupid) compiler takes two milliseconds to compile

a bytecode and speeds up the execution time from one millisecond for interpretation

to half a millisecond. Then, $T_C = 2ms$, $T_I = 1ms$, $T_E = 0.5ms$. So

$x_{opt} = \frac{2ms}{1ms - 0.5ms} = 4$, and the compiler should be used on any bytecode executing

four times or more. On the other hand, a more sophisticated compiler may take

- 13 -

12,000 cycles to compile a bytecode, reducing the interpretation time of 1,000 cycles to an execution time of 2 cycles. Then, $T_C = 12,000$, $T_I = 1,000$, $T_E = 2$, and

$$x_{opt} = \frac{12,000}{1,000 - 2} \cong 12.$$ So any bytecode that runs more than 12 times (the answer is slightly greater than 12) should be compiled. Although the former compiler will be used on more bytecodes, it may not yield the best performance. In order to understand this better, we next examine the components of the speedup.

Total speedup from compilation (with perfect knowledge)

[1048]   Armed with the above result for the optimal compilation threshold, we can examine the speedup obtained with the optimal policy. Recall that

$$T_I E(0) - T_{tot}(x)\big|_{x=x_{opt}} = (T_I - T_E)\left[E(\beta) - \beta \overline{F}(\beta)\right]$$

[1049]   The first factor is merely the per-bytecode efficacy of the compiler. Regarding the second factor, we can substitute the definitions of E and F and simplify to obtain:

$$E(\beta) - \beta \overline{F}(\beta) = \int_{\beta}^{\infty} (y - \beta) f(y) dy$$

[1050]   This integral can be interpreted as the expected amount by which the bytecode frequency exceeds the break-even frequency for the population of compiled bytecodes. Accordingly, the characteristics of the program finally make themselves felt: a program with more high-frequency bytecodes will have a larger expected excess. Since the speedup is the product of the factors, a compiler that produces better code is guaranteed to deliver a greater speedup, as long as its compile time increases no more than in proportion to its efficacy (i.e., constant break-even point). This is a sufficient, not necessary condition.

Compilation Policy in the Non-Ideal Scenario: Less-Than-Perfect Information

[1051]   In the more typical case, a profile is not available at the outset of execution of a program. A profile may be obtained offline by using a practice run, or dynamically by instrumenting a virtual machine, for example. A virtual machine can observe the program's behavior as it unfolds in order to predict the future. One of the simplest ways to do this would be to count the number of times each bytecode

- 14 -

executes, and to compile a bytecode when its count surpasses a threshold. We turn now to an analysis of this plausible heuristic and will seek to understand what the optimal compilation threshold might be. Once a profile is obtained, the hazard rate function can be computed, and the compilation threshold may be set at a point where the compiler's break-even point is equal to the reciprocal of the hazard rate. As before, we denote threshold as x.

[1052]   As a side note, we know that if at any time a bytecode will run β more times, it will be beneficial to recompile that bytecode. So, we can expect that the optimal compilation threshold will turn out to be the one that predicts this, and naively might expect it to be β.

[1053]   In this scenario, for any given compilation threshold, the system compiles the same set of bytecodes as before, but it compiles them later. Any bytecode that eventually gets compiled must first slog through x interpreted executions instead of breezing through x compiled executions. So the total execution time will be the same as it was with perfect knowledge, plus a penalty term of $(T_I - T_E)x\overline{F}(x)$, the product of the additional time spent per execution, the number of executions before compilation, and the number of bytecodes involved:

$$T_{tot\,Im\,perfect}(x) = T_{totPerfect}(x) + (T_I - T_E)x\overline{F}(x)$$
$$= T_I E(0) - (T_I - T_E)[E(x) - \beta\overline{F}(x)] + (T_I - T_E)x\overline{F}(x)$$
$$= T_I E(0) - (T_I - T_E)[E(x) - (x + \beta)\overline{F}(x)]$$

[1054]   As before, it has been rearranged into a term representing the time taken by a totally interpreted run, $T_I E(0)$ and a term representing the speedup thanks to the compiler. Also, as before, we denote the speedup normalized by the per-bytecode execution savings:

$$Speedup_{imperfect}(x) = \frac{T_I E(0) - T_{tot\,Im\,perfect}(x)}{T_I - T_E}$$
$$= E(x) - (x + \beta)\overline{F}(x)$$

[1055]   Comparing this with the perfect knowledge speedup, $E(x) - \beta\overline{F}(x)$, we can observe that the price of ignorance is $x\overline{F}(x)$.

- 15 -

Optimal compilation strategy with imperfect knowledge

[1056] To find the optimal threshold where the speedup will be maximized, we start by differentiating and substituting for $\overline{F}$ and E :

$$
\begin{aligned}
Speedup'_{imperfect}(x) &= E'(x) - (x + \beta)\overline{F}'(x) - \overline{F}(x) \\
&= -xf(x) + xf(x) + \beta f(x) - \overline{F}(x) \\
&= \beta f(x) - \overline{F}(x)
\end{aligned}
$$

[1057] We will also need the second derivative:

$$
Speedup''_{imperfect}(x) = \beta f'(x) + f(x)
$$

[1058] At maximal speedup (recalling that β is always positive for the third step):

$$
\left\{ Speedup'_{imperfect}(x) \Big|_{x=x_{opt}} = 0 \right\} \wedge \left\{ Speedup''_{imperfect}(x) \Big|_{x=x_{opt}} < 0 \right\}
$$

$$
\left\{ \beta f(x_{opt}) - \overline{F}(x_{opt}) = 0 \right\} \wedge \left\{ \beta f'(x_{opt}) + f(x)_{opt} < 0 \right\}
$$

$$
\left\{ \frac{f(x_{opt})}{\overline{F}(x_{opt})} = \frac{1}{\beta} \right\} \wedge \left\{ f'(x_{opt}) < \frac{-f(x)_{opt}}{\beta} \right\}
$$

[1059] At this point, the field of reliability engineering, a field used to study the reliability of light bulbs, semiconductors and the like. (See, e.g., Richard E. Barlow and Frank Proschan, Mathematical Theory of Reliability, SIAM Classics in Applied Mathematics, Philadelphia, 1996, originally published by John Wiley & Sons, Inc., New York, 1965.) For any probability density $f(x)$, the *hazard rate* (or failure rate) is defined as:

$$
hr(x) = \frac{f(x)}{\overline{F}(x)}
$$

[1060] Thus the requirement for optimality is just:

$$
\left\{ hr(x) = \frac{1}{\beta} \right\} \wedge \left\{ f'(x_{opt}) < \frac{-f(x)_{opt}}{\beta} \right\}
$$

[1061] In reliability engineering, the hazard rate can be interpreted as the probability that a part will fail, given that it has operated without failure for x time units. In our case, we can interpret the hazard rate as the probability that a bytecode will stop being executed, given that it has already run x times. It makes sense that when the hazard rate is too high we don't want to compile the bytecode because it is too likely to stop being executed. Likewise, when the hazard rate is too low, we are

- 16 -

too late in compiling the bytecode, it is so unlikely to stop being executed that it should have been compiled before.

[1062] Thus, we can hypothesize that an optimal threshold will exist only in a region where the hazard rate is falling. Let's confirm this:

5

$$hr'(x) = \frac{d}{dx}\left[\frac{f(x)}{\overline{F}(x)}\right]$$

$$= \frac{f'(x)}{\overline{F}(x)} - \frac{f(x)}{\overline{F}(x)^2}\overline{F}'(x)$$

$$= \frac{f'(x)}{\overline{F}(x)} - \frac{f(x)^2}{\overline{F}(x)^2}$$

$$hr'(x)\Big|_{x=x_{opt}} = \frac{f'(x_{opt})}{\overline{F}(x_{opt})} + \frac{f(x_{opt})^2}{\overline{F}(x_{opt})^2}$$

[1063] Substituting in the optimality conditions

10

$$\left\{hr(x_{opt}) = \frac{1}{\beta}\right\} \wedge \left\{f'(x_{opt}) < \frac{-f(x)_{opt}}{\beta}\right\}$$

$$hr'(x)\Big|_{x=x_{opt}} = \frac{f'(x_{opt})}{\overline{F}(x_{opt})} + \frac{1}{\beta^2} \quad \text{(using the first condition above)}$$

$$hr'(x)\Big|_{x=x_{opt}} < \frac{-f(x)_{opt}}{\beta\overline{F}(x_{opt})} + \frac{1}{\beta^2} \quad \text{(using the second condition)}$$

$$hr'(x)\Big|_{x=x_{opt}} < -\frac{1}{\beta^2} + \frac{1}{\beta^2} \quad \text{(using the first again)}$$

$$hr'(x)\Big|_{x=x_{opt}} < 0$$

15 [1064] Stating the full optimality conditions in terms of the hazard rate:

$$\left[hr(x_{opt}) = \frac{1}{\beta}\right] \wedge \left[hr'(x_{opt}) < 0\right]$$

[1065] Thus, the optimal number of times a bytecode should be interpreted before compiling it will be the frequency where the hazard rate falls through the reciprocal of the break-even point. If the frequency distribution does not exhibit a falling hazard

20 rate, there will be no such optimum; instead it will be better to either compile everything first, or to compile nothing ever.

[1066]   When thinking about this problem, one imagines that a bytecode executes, its counter trips (i.e., the bytecode is observed to execute x times), and the system must decide whether or not to compile it. One may think that the goal is to predict how many more times the bytecode will run given that knowledge that it has run for x times, and that the best value for the compilation threshold will be the one that maximized the expected number of future executions of the bytecode. The reality is slightly more complicated.

[1067]   In order to compute the expected number of additional executions of a bytecode, we will need a formula that gives the probability that the total number of executions of a bytecode is given that the bytecode executes at least x times:

$$f(y|y \geq x) \equiv \text{prob. total executions is } y \text{ given } y \text{ at least } x$$

$$f(y|y \geq x) = \frac{f(y)}{\overline{F}(x)} (given \; y \geq x)$$

[1068]   Since we are interested in the expected number of additional executions, we must integrate and subtract:

$$x_{additional} \equiv y - x$$

$$\langle x_{additional} \rangle = \langle y - x | y \geq x \rangle$$

$$\langle x_{additional} \rangle = \int_x^\infty (y - x) \frac{f(y)}{\overline{F}(x)} dy$$

$$\langle x_{additional} \rangle = \frac{\int_x^\infty y f(y) dy}{\overline{F}(x)} - \frac{\int_x^\infty x f(y) dy}{\overline{F}(x)}$$

$$\langle x_{additional} \rangle = \frac{E(x)}{\overline{F}(x)} - x$$

[1069]   To maximize the expected number of additional executions (given that a bytecode has executed x times) this is what we would be optimizing. However, we need to factor in the cost of compilation. The additional number of executions determines the benefit from compilation, but what we really want to optimize is the benefit minus the cost, and (for our single bytecode) this is $\langle x_{additional} \rangle - \beta$.

Furthermore, instead of optimizing for a single bytecode, we need to optimize for all of them. So, we must multiply this expression (the net benefit for a single bytecode)

- 18 -

by the number of applicable bytecodes. But this is just $\overline{F}(x)$. Putting it all together, we want to optimize $\left(\langle x_{additional}\rangle - \beta\right) \times \overline{F}(x)$:

$$\left(\langle x_{additional}\rangle - \beta\right) \times \overline{F}(x) = \left[\frac{E(x)}{\overline{F}(x)} - x - \beta\right] \times \overline{F}(x)$$
$$= E(x) - (x + \beta)\overline{F}(x)$$

5   [1070]   This is the same expression we optimized based on speedup instead of expected additional executions. The two really are equivalent.

Results with analytic frequency distributions

[1071]   Armed with the above analytical results, we now examine two simple continuous distributions. First, assume that the bytecode frequency is uniformly

10   distributed up to 1000 (unrealistic perhaps, but a useful example in its simplicity). Figures 3A through 3D respectively show plots of the bytecode frequency density $f(x)$, the complementary cumulative distribution $\overline{F}(x)$, hazard rate, and speedups, assuming a break-even point of 100. Recall that the density shows how many bytecodes have *exactly* a given number of repetitions, and that the complementary

15   cumulative distribution shows how many bytecodes have *at least* a given number of repetitions. As expected, the perfect speedup peaks at 100, and the imperfect one has no optimum, except for compiling everything right away. (Of course, there is no profit in compiling any bytecode with a frequency less than one, a distortion which occurs due to the use of integrals instead of summations.)

20   [1072]   Next, we turn to a distribution with a decreasing hazard rate, the log normal distribution. This distribution assumes that the log of the frequencies is normally distributed (a more realistic assumption than the uniform distribution shown in Figure 3A). Figures 4A through 3E respectively show plots of the density, complementary distribution, hazard rate, hazard rate with tail, and speedups. Since we have made no

25   attempt to calibrate the parameters of the distribution, the scale of the graphs below has no significance. For example, the break-even point was set to 1.4 for these plots. Recall that the optimal compilation threshold given imperfect knowledge occurs where the hazard rate decreases through $\frac{1}{\beta}$. In this distribution with a falling hazard

rate, the imperfect speedup does display a peak, right where the hazard rate falls through $\frac{1}{\beta} = 0.7$ (on the y-axis of Figure 4C), which corresponds to a compilation threshold of just less than 2 (on the x-axis of Figure 4C). Therefore, an optimal compilation policy (according to our model) would be to compile any bytecode after

5   it executes two times. Note that in a real life example, the compilation threshold may be much higher. Also, in real life, the hazard rate would likely begin increasing to form a typical hazard rate "U" curve if the program has a finite lifetime. Such "right halves" of the hazard rate curve will be positioned differently, but they do not affect the optimal compilation threshold calculation in any way.

10   [1073]   One may guess what the optimal imperfect-knowledge compilation threshold might be, since it might be necessary to observe β executions before we are willing to bet on β additional executions. If that were the case, then, since the optimal threshold occurs where the hazard rate equals the reciprocal of β, we would expect that $hr(x) = \frac{1}{x}$, and our plots would be straight lines with slopes of -1. Now,

15   the right hand sides of our curves must have positive slope because our programs have finite lifetimes. In other words, as approaches the maximum for the program, the probability that the bytecode will cease to be executed must increase. And on the left hand side of real life programs, the curves typically fall more steeply than log-log hyperbolae. Thus, the optimal compilation threshold typically lies at a fraction of the

20   break-even point, and it may be too conservative to wait until a bytecode's frequency reaches β to compile it.

[1074]   Also, there may be a limit to the optimizations that a dynamic compiler may perform. Recall that $\beta = \frac{T_C}{T_I - T_E}$. As the compiler gets more sophisticated, the T_I term will dominate the denominator, and since T_C will be rising, β will have to rise

25   eventually. However, typical plots show that if β goes much above 500, then since the hazard rates never drop that low, the model predicts there will be no optimal compilation threshold. With such a compiler, the best strategy would be to either compile all or nothing at all.

[1075]   When the compilation threshold is low (approaching unity), it is possible to squander half of the available speedup (e.g. for pep test) on premature optimization. In fact, many commercially available virtual machines with JITs do just this, compiling any code that executes even one time. Increasing the compilation threshold

5   to as little as two would recover most of the available speedup.

Revisiting assumptions

[1076]   In system 100, any bytecode can be compiled independently of any other. In systems where whole methods are compiled at a time, the break-even points may be expected to be worse, since the net effect will be to increase the compile time

10   above that required to merely compile an inner loop. However, the basic model will likely still hold. For example when one computes the compilation speed per bytecode for the above described embodiment, do not divide the total compiler time by the number of bytecodes compiled. Instead, divide the total compiler time by the number of bytecodes executed by the program (i.e., the sum of all the bytecodes in the

15   interpreter histogram). That gives a more accurate break-even point. Even if a variation in the amount of dead or non-loop code in each method will tend to fuzz the results, the model is still effective.

[1077]   The foregoing discussion assumes that compilation speeds up the execution of each bytecode by the same amount. Of course, the amount of speedup will vary

20   from bytecode to bytecode. As long as this variation is independent of frequency, the effect should merely be to add noise to the predictions.

[1078]   Finally the model assumes that the preparation unit 120 can count the executions of each bytecode. This data may be approximated in real life by less accurate, more efficient means such as strategically placed counters.

25   [1079]   A simple analytical model can illuminate the relationship of dynamic compilation policy to the total running time of a program. Given a bytecoded program with bytecode frequency distribution $f(x)$, and an implementation whose compiler has a break-even point , it is possible to compute an optimal compilation threshold. If profile information is available, it is best to compile every bytecode

30   whose frequency exceeds β . Without profile information, if a bytecode is compiled

- 21 -

after it has executed x times, it is best to set x where the hazard rate, $\dfrac{f(x)}{F(x)}$ falls

through $\dfrac{1}{\beta}$. A term taken from reliability engineering, the hazard rate can be

interpreted as the probability that a bytecode will not be executed any more after it has already run times.

5   [1080]   The analytical model we have described can be used to evaluate multiple stages of dynamic compilation. Also, when deciding whether it would be advantageous to slow the compiler in order to improve code quality, these results provide a sufficient condition: if β does not increase, the slower compiler that produces better code will increase system performance. Also, the performance of a

10   system that compiles every bytecode that runs can be improved by delaying compilation until a bytecode has run at least twice. However, if it takes more than about 500 executions to recoup the compilation time, there may be no optimal compilation threshold.

[1081]   The above description is intended to describe at least one embodiment of

15   the invention, not to define the scope of the invention. Rather, the scope of the invention is defined in the claims below. Thus, other embodiments of the invention include other variations, modifications, additions, and/or improvements to the above description.

[1082]   Those skilled in the art will recognize that boundaries between logic blocks

20   are merely illustrative and that alternative embodiments may merge logic blocks or circuit elements or impose an alternate decomposition of functionality upon various logic blocks or circuit elements. Moreover, alternative embodiments may combine multiple instances of a particular component.

[1083]   System 100, and any portions thereof, may be integrated on-chip, on-board,

25   in-box or may be distributed. System 100 may be hardware or software or both. For example, the architectural blocks may be integrated circuits on a board, software implementations for execution on general purpose processors, a combination thereof, a virtual machine, or software representations of hardware for design and incorporation into other circuitry.

[1084]   In one embodiment, system 100 is a computer system such as a personal computer system. Other embodiments may include different types of computer systems. Computer systems are information handling systems which can be designed to give independent computing power to one or more users. Computer systems may

5   be found in many forms including but not limited to mainframes, minicomputers, servers, workstations, personal computers, notepads, personal digital assistants, various wireless devices and embedded systems. A typical computer system includes at least one processing unit, associated memory and a number of input/output (I/O) devices.

10   [1085]   A computer system processes information according to a program and produces resultant output information via I/O devices. A program is a list of instructions such as a particular application program and/or an operating system. A computer program is typically stored internally on computer readable storage medium or transmitted to the computer system via a computer readable transmission medium.

15   A computer process typically includes an executing (running) program or portion of a program, current program values and state information, and the resources used by the operating system to manage the execution of the process. A parent process may spawn other, child processes to help perform the overall functionality of the parent process. Because the parent process specifically spawns the child processes to

20   perform a portion of the overall functionality of the parent process, the functions performed by child processes (and grandchild processes, etc.) may sometimes be described as being performed by the parent process.

[1086]   A personal computer system can usually be defined as a desk top, floor standing, or portable microcomputer that includes a system unit having a system

25   processor and associated volatile and nonvolatile memory, a display monitor, a keyboard, one or more diskette drives, a fixed disk storage device and an optional printer. A system board is often used to electrically connect these components together. A personal computer system may also include one or a plurality of I/O devices (i.e. peripheral devices) which are coupled to the system processor and which

30   perform specialized functions. Examples of I/O devices include modems, sound and video devices or specialized communication devices. Mass storage devices such as

hard disks, CD-ROM drives and magneto-optical drives are also considered to be peripheral devices.

[1087] Each of the blocks/steps of Figure 2 may be executed by a module (e.g., a software module) or a portion of a module or a computer system user. Thus, the above described method(s), the steps thereof and modules therefor may be executed on a computer system configured to execute the operations of the method and/or may be executed from computer-readable media. The method and/or modules may be embodied in a machine-readable and/or computer-readable medium for configuring a computer system to execute the method. Thus, the software modules may be stored within and/or transmitted to a computer system memory to configure the computer system to perform the functions of the module. Software modules may include script, batch or other executable files, or combinations and/or portions of such files.

[1088] Those skilled in the art will recognize that boundaries between the functionality of the above described steps merely illustrative. The functionality of multiple steps may be combined into a single step, and/or the functionality of a single step may be distributed in additional steps. Moreover, alternative embodiments may include multiple instances of a particular step, and the order of steps may be altered in various other embodiments. Likewise, those skilled in the art will recognize that boundaries between modules are merely illustrative and alternative embodiments may merge modules or impose an alternative decomposition of functionality of modules. For example, the modules discussed herein may be decomposed into submodules to be executed as multiple computer processes. Moreover, alternative embodiments may combine multiple instances of a particular module or submodule.

[1089] Software modules may be received by system 100, for example, from computer readable media such as storage 110. Computer readable media may be permanently, removably or remotely coupled to system 100. Computer readable media may include, for example and without limitation, any number of the following: magnetic storage media including disk and tape storage media; optical storage media such as compact disk media (e.g., CD-ROM, CD-R, etc.) and digital video disk storage media; nonvolatile memory storage media including semiconductor-based memory units such as FLASH memory, EEPROM, EPROM, ROM; ferromagnetic

- 24 -

digital memories; holographic media, volatile storage media including registers, buffers or caches, main memory, RAM, etc.; and data transmission media including computer networks, point-to-point telecommunication equipment, and carrier wave transmission media, just to name a few. Other new and various types of computer-

5   readable media may be used to store and/or transmit the software modules discussed herein.

[1090]   It is to be understood that the architectures depicted herein are merely exemplary, and that in fact many other architectures can be implemented which achieve the same functionality. In an abstract, but still definite sense, any

10   arrangement of components to achieve the same functionality is effectively "associated" such that the desired functionality is achieved. Hence, any two components herein combined to achieve a particular functionality can be seen as "associated with" each other such that the desired functionality is achieved, irrespective of architectures or intermedial components. Likewise, any two

15   components so associated can also be viewed as being "operably connected", or "operably coupled", to each other to achieve the desired functionality.

[1091]   The components and devices described herein are used as examples for sake of conceptual clarity. Consequently, as used herein these specific exemplars are intended to be representative of their more general classes. Furthermore, in general,

20   the use of any specific exemplar herein is also intended to be representative of its class and the noninclusion of any specific devices in any exemplary lists herein should not be taken as indicating that limitation is desired.

[1092]   The above detailed description has been divided into sections with subheadings in order to highlight the invention described herein; however, those

25   skilled in the art will appreciate that such sections are merely for illustrative focus, and that the invention herein disclosed typically draws its support from multiple sections. Consequently, it is to be understood that the division of the detailed description into separate sections is merely done as an aid to understanding and is in no way intended to be limiting.

30   [1093]   Because the above detailed description is exemplary, when "one embodiment" is described, it is an exemplary embodiment. Accordingly, the use of

- 25 -

the word "one" in this context is not intended to indicate that one and only one embodiment may have a described feature. Rather, many other embodiments may, and often do, have the described feature of the exemplary "one embodiment." Thus, as used above, when the invention is described in the context of one embodiment, that

5 one embodiment is one of many possible embodiments of the invention.

[1094]   Notwithstanding the above caveat regarding the use of the words "one embodiment" in the detailed description, it will be understood by those within the art that if a specific number of an introduced claim element is intended in the below claims, such an intent will be explicitly recited in the claim, and in the absence of

10 such recitation no such limitation is present or intended. For example, in the claims below, when a claim element is described as having "one" feature, it is intended that the element be limited to one and only one of the feature described. Furthermore, when a claim element is described in the claims below as including or comprising "a" feature, it is not intended that the element be limited to one and only one of the feature

15 described. Rather, for example, the claim including "a" feature reads upon an apparatus or method including one or more of the feature in question. That is, because the apparatus or method in question includes a feature, the claim reads on the apparatus or method regardless of whether the apparatus or method includes another such similar feature. This use of the word "a" as a nonlimiting, introductory article to

20 a feature of a claim is adopted herein by Applicants as being identical to the interpretation adopted by many courts in the past, notwithstanding any anomalous or precedential case law to the contrary that may be found. Similarly, when a claim element is described in the claims below as including or comprising an aforementioned feature (e.g., "the" feature), it is intended that the element not be

25 limited to one and only one of the feature described merely by the incidental use of the definite article.

[1095]   Furthermore, the use of introductory phrases such as "at least one" and "one or more" in the claims should not be construed to imply that the introduction of another claim element by the indefinite articles "a" or "an" limits any particular claim

30 containing such introduced claim element to inventions containing only one such element, even when the same claim includes the introductory phrases "one or more"

- 26 -

or "at least one" and indefinite articles such as "a" or "an." The same holds true for the use of definite articles.

[1096]   While particular embodiments of the present invention have been shown and described, it will be obvious to those skilled in the art that, based upon the teachings herein, various modifications, alternative constructions, and equivalents may be used without departing from the invention claimed herein. Consequently, the appended claims encompass within their scope all such changes, modifications, etc. as are within the true spirit and scope of the invention. Furthermore, it is to be understood that the invention is solely defined by the appended claims. The above description is not intended to present an exhaustive list of embodiments of the invention. Unless expressly stated otherwise, each example presented herein is a nonlimiting or nonexclusive example, whether or not the terms nonlimiting, nonexclusive or similar terms are contemporaneously expressed with each example. Although an attempt has been made to outline some exemplary embodiments and exemplary variations thereto, other embodiments and/or variations are within the scope of the invention as defined in the claims below.